Decision Procedures for the Formal Analysis of Software

David Déharbe[1,⋆], Pascal Fontaine[2],
Silvio Ranise[2,3], and Christophe Ringeissen[2]

[1] UFRN/DIMAp, Natal, Brazil
[2] LORIA, Nancy, France
[3] Univerisità di Milano, Italy
`david@dimap.ufrn.br, fontaine@loria.fr, ranise@loria.fr, ringeiss@loria.fr`

1 Introduction

Catching bugs in programs is difficult and time-consuming. The effort of debugging and proving correct even small units of code can surpass the effort of programming. Bugs inserted while "programming in the small" can have dramatic consequences for the consistency of a whole software system as shown, e.g., by viruses which can spread by exploiting buffer overflows, a bug which typically arises while coding a small portion of code. To detect this kind of errors, many verification techniques have been put forward such as static analysis and model checking.

Recently, in the program verification community, there seems to be a growing demand for more declarative approaches in order to make the results of the analysis readily available to the end user.[1] To meet this requirement, a growing number of program verification tools integrate some form of theorem proving.

The goals of our research are two. First, we perform theoretical investigations of various combinations of propositional and first-order satisfiability checking so to automate the theorem proving activity required to solve a large class of program analysis problems which can be encoded as first-order formulae. Second, we experimentally investigate how our techniques behave on real problems so to make program analysis more precise and scalable. Building tools capable of providing a good balance between precision and scalability is one of the crucial challenge to transfer theorem proving technology to the industrial domains.

2 Designing Decision Procedures

Decision procedures, their combination, and their integration with other reasoning activities (such as Boolean solving or quantifier handling) have recently

⋆ This author has been supported in part by CNPq grant 506469/04-2.

[1] See, for example, the challenge at `http://research.microsoft.com/specncheck/consel_challenge.htm`

attracted a lot of attention because of their importance for many verification techniques (such as bounded model checking, software model checking, and deductive verification to name but a few). In this tutorial, we will describe some of the techniques which allow us to build, combine, and integrate decision procedures.

2.1 Building

A lot of papers in the literature address the problem of building decision procedure for theories of interest in program verification, such as [8]. The methods used in these papers are rather *ad hoc* and seem difficult to generalize.

We will present the so-called rewriting approach to decision procedures [2] for theories which can be axiomatized by a finite set of clauses (in first-order logic with equality) which are quite relevant for software verification: the theory of uninterpreted function symbols, theories of (possibly cyclic) lists, the theory of arrays (with or without extensionality), and their combinations. This approach allows us to synthesize such procedures in a uniform way by working in a well-understood framework for all the theories listed above. The proof that the decision procedures are correct is straightforward w.r.t. other correctness proofs given in the literature since it amounts to proving the termination of the exhaustive application of the rules of a calculus (see [2] for details). Furthermore, these theoretical results pave the way to synthesizing decision procedures from rewriting-based theorem provers (almost) taken off-the-shelf. We will present experimental results [1] which confirm the practical feasibility of this approach by showing that an automated theorem prover compares favorably with *ad hoc* decision procedures.

2.2 Combining

Verification problems frequently require more than just a single theory to model a given system and/or specify a property that we would like the system to satisfy. Hence, there is an obvious need to combine the decision procedures which are available for some component theories in a modular way so to obtain a decision procedure for unions of theories. This modular approach is particularly interesting to combine (fragments of) Presburger Arithmetics (for which the rewriting-based approach does not work) with rewriting-based decision procedures. There has been a long series of works devoted to the combination of decision procedures in the context of program verification. This line of research was started in the early 80's by two combination schemas independently presented by Nelson-Oppen [9] and Shostak [14] for unions of theories with disjoint signatures. Recently, a series of papers have clarified the connections between both combination schemas [13].

We will present a rational reconstruction of combination schemas [11] which will allow us to derive and prove correct Nelson-Oppen and Shostak combination schemas in a simple and uniform way. The reconstruction is based on a classification of the semantic properties that the theories being combined should satisfy (e.g., being stably-infinite). Then, we describe how some of the schemas might be generalized in order to find a better trade-off between the simplicity of Nelson-Oppen schema and the efficiency of Shostak's. We will discuss how to lift some of the requirements needed for the Nelson-Oppen combination schema to work; e.g., both theories need to be stably-infinite [12]. This is particularly relevant to software verification problems involving container data structures (such as lists, arrays, or sets) and the elements stored in such data structures whose theories may not satisfy the requirement of being stably-infinite (consider, for example, enumerated data-types). Finally, we will explain how rewriting-based procedures can be efficiently combined with arbitrary decision procedures in the Nelson-Oppen schema by showing that, under suitable assumptions, they derive all facts that need to be exchanged for the synchronization of the states of the procedures [7].

2.3 Integrating

When building decision procedures for certain theories or unions of theories, only the problem of checking the satisfiability of conjunctions of literals is considered. Now, verification problems often generate proof obligations consisting of complex Boolean combination of ground literals and may even contain quantifiers. So, to make the decision procedures really usable for software verification, it is crucial to integrate them with (*i*) Boolean solvers (such as SAT solvers or BDDs) and with (*ii*) mechanisms to handle quantifiers. Such system are called Satisfiability Modulo Theory solvers. The idea underlying (*i*) is to consider a propositional abstraction of the formula to be checked for satisfiability and then enumerating its propositional assignments. Such assignments are then refined back to conjunctions of ground literals which are checked for satisfiability by means of an available decision procedure. If all the (refined) propositional assignments are discarded as unsatisfiable with respect to the theory, we can conclude that the original formula is unsatisfiable. Otherwise, the formula is satisfiable. This is a very hot topic in automated deduction and verification as witnessed by many systems based on this type of integration.[2] The idea underlying (*ii*) is to preprocess the formula in order to abstract away the quantified sub-formulas by propositional letters and, at the same time, to enrich the background theory with enough information for a first-order theorem prover to refine the abstraction. In this way, we obtain a ground formula which must be checked for satisfiability modulo an extended theory. If the decision procedure can cope with the extended theory, it is possible to use (*i*) in order to solve the new satisfiability problem. We will discuss the encouraging experimental results obtained with an implementation of such techniques (see Section 3 for more details) on a set of benchmarks taken from the certification of auto-generated aerospace code [4].

[2] See the Satisfiability Modulo Theory Library at `http://combination.cs.uiowa.edu/smtlib` for pointers to the available systems.

2.4 Embedding

Formal system verification calls for expressive specification languages, but also requires highly automated tools. These two goals are not easy to reconcile, especially if one also aims at high assurances for correctness. Interactive proof assistants encode rich logics, which are at the basis of highly expressive (and user-extensible) modeling languages. Their verification environment is often built around a small kernel that ensures that theorems can only be produced from given axioms and proof rules; this approach helps to keep the trusted code base small and therefore gives high assurance of correctness. These tools however do not focus on automation, and much interaction is often required for even simple (but tedious) reasoning. At the other end of the spectrum one finds decision procedures, based on a restricted language, but that provide fully automatic (and efficient) deductive capabilities within that language.

There is a growing interest in making interactive proof assistants and automatic tools based on decision procedures cooperate in a safe way. This allows assistants to delegate proofs of formulas that fall within the scope of automatic tools. First, this involves translating formulas from the language of the assistant to the language of the automatic tool. Second, to comfort the confidence in the translation process and in the automatic tool, it is necessary to extract a proof from the automatic tool and certify it within the trusted kernel of the proof assistant. We will focus on proof extraction from decision procedures, but also mention state-of-the-art techniques for general first-order automatic theorem provers, and investigate proof certification for proof assistants. In particular, we will examine our recent [6] and ongoing work on combining the system **haRVey** (see Section 3) with the Isabelle [10] proof assistant.

3 Implementing Decision Procedures: haRVey

All the techniques discussed in Section 2 are implemented (or being implemented) in a system, called **haRVey**[3]. By now, the system has two incarnations. The former (called **haRVey-FOL**) integrates Boolean solvers with an automated theorem prover, to implement the rewriting-based decision procedures overviewed in Section 2.1 (see [3,4]). The latter (called **haRVey-SAT**) integrates Boolean solvers with a combination of decision procedures for the theory of uninterpreted function symbols and Linear Arithmetic based on the Nelson-Oppen schema and techniques to handle quantifiers and lambda-expressions (see [5]). Furthermore, **haRVey-SAT** can produce proofs which can then be independently checked by the proof assistant Isabelle (see [6]).

While **haRVey-FOL** offers a high degree of flexibility and automation for a variety of theories, **haRVey-SAT** is usually faster on problems with simpler background theories and ensures a high degree of certification by its proof checking capability. Along the lines hinted in [7], our current work aims at merging the two incarnations in one system which retains the flexibility and high-degree of

[3] `http://harvey.loria.fr/`

automation for expressive theories of **haRVey-FOL** and provides better performances on simpler problems as **haRVey-SAT**.

References

1. A. Armando, M. P. Bonacina, S. Ranise, and S. Schulz. On a rewriting approach to satisfiability procedures: extension, combination of theories and an experimental appraisal. In B. Gramlich, editor, *Frontiers of Combining Systems (FroCoS)*, volume 3717 of *Lecture Notes in Computer Science*, pages 65–80. Springer, 2005.
2. A. Armando, S. Ranise, and M. Rusinowitch. A Rewriting Approach to Satisfiability Procedures. *Information and Computation*, 183(2):140–164, June 2003.
3. D. Déharbe and S. Ranise. Light-Weight Theorem Proving for Debugging and Verifying Units of Code. In *Proc. of the Int. Conf. on Software Engineering and Formal Methods (SEFM03)*, pages 220–228. IEEE Computer Society, 2003.
4. D. Déharbe and S. Ranise. Satisfiability Solving for Software Verification. In *Proc. of IEEE/NASA Workshop on Leveraging Applications of Formal Methods, Verification, and Validation (ISoLA'05)*, 2005.
5. P. Fontaine. *Techniques for verification of concurrent systems with invariants.* PhD thesis, Institut Montefiore, Université de Liège, Belgium, Sept. 2004.
6. P. Fontaine, J.-Y. Marion, S. Merz, L. P. Nieto, and A. Tiu. Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 3920 of *LNCS*, pages 167–181. Springer, 2006.
7. H. Kirchner, S. Ranise, C. Ringeissen, and D.-K. Tran. On Superposition-Based Satisfiability Procedures and their Combination. In *International Conference on Theoretical Aspects of Computing (ICTAC)*, volume 3722 of *Lecture Notes in Computer Science*, pages 594–608. Springer, 2005.
8. G. Nelson. Techniques for Program Verification. Technical Report CSL-81-10, Xerox Palo Alto Research Center, June 1981.
9. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. on Programming Languages and Systems*, 1(2):245–257, Oct. 1979.
10. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic.* Number 2283 in Lecture Notes in Computer Science. Springer-Verlag, 2002.
11. S. Ranise, C. Ringeissen, and D.-K. Tran. Nelson-Oppen, Shostak and the Extended Canonizer: A Family Picture with a Newborn. In Z. Liu and K. Araki, editors, *International Conference on Theoretical Aspects of Computing (ICTAC)*, volume 3407 of *Lecture Notes in Computer Science*, pages 372–386. Springer, 2005.
12. S. Ranise, C. Ringeissen, and C. G. Zarba. Combining data structures with nonstably infinite theories using many-sorted logic. In B. Gramlich, editor, *Frontiers of Combining Systems (FroCoS)*, volume 3717 of *Lecture Notes in Computer Science*, pages 48–64. Springer, 2005.
13. N. Shankar and H. Rueß. Combining Shostak theories. In S. Tison, editor, *Proc. of the 13th Int. Conf. on Rewriting Techniques and Applications*, volume 2378 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2002.
14. R. E. Shostak. Deciding combinations of theories. *J. of the ACM*, 31:1–12, 1984.